Intermediate OOP in Drupal

Patterns, Services, Events, and Dependency Injection

Presented by Jonathan Daggerhart of Daggerhart Lab

Who am I?

Jonathan Daggerhart

Lead Developer at Daggerhart Lab
Organizer of Drupal Camp Asheville

Drupal: daggerhart
WordPress: daggerhart
Twitter: @daggerhart
YouTube: /c/JonathanDaggerhart

daggerhartlab.com

Intermediate OOP in Drupal

Overview

1. Terminology
2. Concepts
3. Useful OOP Patterns
4. Drupal / Symfony
5. Recap

Terminology - Class Basics

Term	Definition
Interface	Template (aka, contract) for a class. Classes "implement" interfaces
Class	Template for an object that defines the properties and methods of an object
Object	An instance of a class

```php
interface PizzaInterface {}

class Pizza implements PizzaInterface {}

$pizza1 = new Pizza();
$pizza2 = new Pizza();
```

Terminology - Interfaces

Defines required method signatures of a class

```php
interface PizzaInterface {

  /**
   * Add a topping to the pizza.
   *
   * @param string $topping
   */
  public function addTopping($topping);

  /**
   * Get all toppings on the pizza.
   *
   * @return array
   */
  public function getToppings();

}
```

Terminology - Classes

Provides implementation details of the methods defined in interfaces.

Classes "implement" interfaces.

```
class Pizza implements PizzaInterface {

  private $toppings = [];

  public function addTopping($topping) {
    $this->toppings[] = $topping;
  }

  public function getToppings() {
    return $this->toppings;
  }

}
```

Terminology - Objects

Objects are unique instances of a Class

```php
$pizza1 = new Pizza();
$pizza1->addTopping('sauce');
$pizza1->addTopping('cheese');
print_r($pizza1->getToppings());
/*
  Array
   0 => sauce
   1 => cheese
*/

$pizza2 = new Pizza();
$pizza2->addTopping('mustard');
print_r($pizza2->getToppings());
/*
  Array
   0 => mustard
*/
```

Terminology - Visibility

What parts of class can be used other parts of the system

Term	Definition
Public	Everything can use and modify public properties and methods
Private	Restricted access to only methods within the class
Protected	Restricted to the class and its descendents

Visibility Example

```php
class Pizza {

  // Only this class and its descendents can access this property.
  protected $toppings = [];

  // Any part of the system can run this method.
  public function getToppings() {
    return $this->toppings;
  }

  // Only this class can run this method.
  private function removeAllToppings() {
    $this->toppings = [];
  }
}
```

```php
$pizza = new Pizza();
$pizza->toppings; // Fatal error
$pizza->removeAllToppings(); // Fatal error
$pizza->getToppings(); // This works!
```

Concepts

Object Basics

Concept	Definition
Data Types	An attribute of data that describes itself to the system Examples: integer, string, float, array, stdClass
Type Hinting	Defining the required type of data required of a method
Inheritance	Deriving a new class from another class or interface
Polymorphism	Any instance of the type of thing is valid

Types & Type Hinting

When creating classes, we are creating new Data Types

```php
// Creating a class defines a new Type - Pizza & PizzaOven
class Pizza {}
```

```php
class PizzaOven {
  // Hint that a Pizza object is the required data type
  public function cook(Pizza $any_pizza) {}
}
```

```php
$pizza = new Pizza();
var_dump(is_a($pizza, 'Pizza')); // outputs "bool(true)"

$oven = new PizzaOven();
var_dump(is_a($oven, 'PizzaOven')); // outputs "bool(true)"

$oven->cook($pizza); // Cool

$not_a_pizza = 1;
$oven->cook($not_a_pizza); // Fatal error!
```

Inheritance - Extending Classes

We can create new classes that are based on other classes by "extend"ing a class

```php
class Pizza {
  public $toppings = [];
  public function addTopping($topping) {
    $this->toppings[] = $topping;
  }
}
```

```php
class MustardPizza extends Pizza {
  public function addMustard() {
    $this->toppings[] = 'mustard';
  }
}
```

```php
$pizza = new MustardPizza();
$pizza->addTopping('cheese');
$pizza->addMustard();

var_dump(is_a($pizza, 'MustardPizza')); // outputs "bool(true)"
var_dump(is_a($pizza, 'Pizza')); // outputs "bool(true)"
```

Inheritance - Interfaces

Implementing interfaces is another form of inheritance

```php
interface PizzaInterface {}
```

```php
class Pizza implements PizzaInterface {}
```

```php
$pizza = new Pizza();

var_dump(is_a($pizza, 'Pizza')); // outputs "bool(true)"
var_dump(is_a($pizza, 'PizzaInterface')); // outputs "bool(true)"
```

Polymorphism

If a parameter expects a certain type, any object that is an instance of that type can be provided as a valid argument.

```php
interface PizzaInterface {}

// Pizza is a PizzaInterface.
class Pizza implements PizzaInterface, AnotherInterface, YetAnother {}

// MustardPizza is a Pizza, and is a PizzaInterface.
class MustardPizza extends Pizza {}

class PizzaOven {
  // Type-hint the interface.
  public function cook(PizzaInterface $pizza) {}
}
```

```php
$specialty_pizza = new MustardPizza();
$oven = new PizzaOven();

// PizzaOven::cook() accepts anything that is a PizzaInterface
$oven->cook($specialty_pizza);

$oven->cook('not a pizza'); // Fatal error!
```

Best Practice Alert!!!

Program to Interfaces

```
class PizzaOven {

  // Don't do this, or we'll never be able to cook our MustardPizza.
  public function cook(Pizza $pizza) {}

  // Do this. Type hint the interface.
  public function cook(PizzaInterface $pizza) {}

}
```

Practical Implication?

1. Write interfaces when creating new custom classes
2. Type-hint interfaces when creating new class methods

Exercise: Feed the Rabbit

Create two new data types: `Rabbit` and `Carrot`

- Each type should implement at least one interface
- Rabbits should be able to `eat()`
- Write a program to feed a Carrot to a Rabbit

Hint: don't over-think it

https://sandbox.onlinephpfunctions.com/

Result: Happy Rabbit

```php
// Food is a new data type.
interface Food {}

// Carrot is a new data type, and is a Food.
class Carrot implements Food {}

// Animal is a new data type.
interface Animal {
  // Animals eat Food.
  public function eat(Food $food);
}

// Rabbit is a new data type, and is an Animal.
class Rabbit implements Animal {
  public function eat(Food $food) {
    echo "Yum yum!";
  }
}
```

```php
$treat = new Carrot();
$bunny = new Rabbit();

$bunny->eat($treat); // Yum yum!
```

Concepts

Object Design Best Practices

Concept	Description
Nouns & Verbs	Two ways to think about what a class should and shouldn't do
Dependencies	When objects depend on other parts of the system to operate successfully
Composition	An approach to building complex objects that encourages flatter data models

Nouns & Verbs

Nouns & Verbs

When creating a new class it's useful to think about the nouns and verbs related to the work.

Nouns

- Have attributes (state), but don't necessarily "do" anything
  (rocks, pictures, books)
- Some perform verbs
  (rabbits Jump, trees Grow, presenters Present)
- Sometimes another thing will perform the verb on the noun
  (people Kick rocks, stores Sell books, ovens Cook pizza)

When designing classes ask yourself,
"What is doing the work on what?"

Nouns & Verbs - cont...

Verbs don't exist in a vacuum, some noun is likely the performing the work.

Consider the follow classes:

```php
class Rock {
  public $weight = 5;
  public $color = 'grey'
}
```

```php
// Catapults launch rocks.
class Catapult {
  public function launch(Rock $rock) {}
}
```

```php
class Rabbit {
  // The state of the Rabbit is that it is either jumping or it isn't.
  protected $isJumping = false;

  public function jump() {
    $this->isJumping = true;
  }
}
```

Nouns & Verbs

Object Design Planning

Before designing classes:

1. Identify the nouns and verbs related to your work.
2. Ask: Does it have a "state"? Can something change about this thing that affects its behavior?
3. Ask: Does/should it perform its verbs itself, or does something else perform the verb on it?

Where reasonable, separate your verbs into classes dedicated to "doing" things.

Nouns & Verbs

Best Practices

Ideally (but not always reasonable) we create two types of classes:

Nouns mutable

A collection of attributes (state).
The state can change, but it doesn't perform actions on other parts of the system.

Do-ers immutable

A collection of verbs that has no state. It performs actions on other objects, and its behavior never changes.

~~Nouns~~ Models

Objects that hold data and optional exposed some behavior for manipulating or retrieving that data

```php
class Book {

  private $title = 'Big Book of Cat Pictures';

  public function getTitle() {
    return $this->title;
  }

}
```

```php
class Person {

  private $bookCollection = [];

  public function addBookToCollection(Book $book) {
    $this->bookCollection[] = $book;
  }

}
```

~~Do-ers~~ Services

Service objects either perform a task or return a piece of information.

Service objects should have no State

```
class Library {

  public function fetchBook(Book $book) {}

  public function shelveBook(Book $book) {}

  public function buyBook(Book $book, Person $supplier) {}

  public function lendBook(Book $book, Person $customer) {}

  public function sellBook(Book $book, Person $customer) {}

}
```

Composition

Assigning an object to another object's property

Building a more complicated object out of simpler objects

Composition

Composing Pizza out of Toppings: Part 1

```php
interface ToppingInterface {
  public function getName(): string;
  public function getPrice(): float;
}
```

```php
class Topping implements ToppingInterface {

  private $name;
  private $price;

  public function __construct(string $topping_name, float $price) {
    $this->name = $topping_name;
    $this->price = $price;
  }

  public function getName() {
    return $this->name;
  }

  public function getPrice() {
    return $this->price;
  }
}
```

Composition

Composing Pizza out of Toppings: Part 2

```php
interface PizzaInterface {
  public function getBasePrice(): float;
  public function getToppings(): array;
  public function addTopping(ToppingInterface $topping);
}
```

```php
class Pizza implements PizzaInterface {

  private $basePrice = 8.50;
  private $toppings = [];

  public function getBasePrice() {
    return $this->basePrice;
  }

  public function getToppings() {
    return $this->toppings;
  }

  public function addTopping(ToppingInterface $topping) {
    $this->toppings[] = $topping;
  }

}
```

Composition

Composing Pizza out of Toppings: Part 3

```
$pizza = new Pizza();
$pizza->addTopping(new Topping('Sauce', 0));
$pizza->addTopping(new Topping('Cheese', 1.50));
$pizza->addTopping(new Topping('Roasted Red Pepper', 2.75));

// Determine the price of this Pizza.
$price = $pizza->getBasePrice();

foreach ($pizza->getToppings() as $topping) {
  $price += $topping->getPrice();
}

echo $price; // 12.75
```

Composition vs Inheritance

1. Both are mechanisms for extending functionality of an object
2. Too much inheritance creates a rigid system
3. Composition is preferred when solving problems outside of the data object's domain

More on this with the Decorator pattern

Dependencies

Requirements a class must have in order to work correctly

Dependencies

There are two main types of dependencies

Configuration Dependencies

- Required for a class to work properly
- Determines the behavior of the class

Contextual Dependencies

- Required for an individual task performed
- Determines the results of the task

Dependencies

Example

Consider this Service that fetches data from an external API

```
class CatApiConsumer {

  private $username = 'daggerhart';
  private $apiKey = 'abc123';
  private $apiBaseUrl = 'https://cat-pictures.api';
  private $httpClient;

  public function __construct() {
    $this->httpClient = new HttpClient([
      'base_url' => $this->apiBaseUrl
    ]);
  }

  public function getRandom() {
    return $this->httpClient->get('/random', [
      'auth' => [ $this->username, $this->apiKey ],
      'query' => [ 'type' => 'tabby' ],
    ]);
  }
}
```

Dependencies

Configuration

- API Credentials
- API Base URL
- Http Client

```
class CatApiConsumer {

  private $username = 'daggerhart';
  private $apiKey = 'abc123';
  private $apiBaseUrl = 'https://cat-pictures.api';
  private $httpClient;

  public function __construct() {
    $this->httpClient = new HttpClient([
      'base_url' => $this->apiBaseUrl
    ]);
  }

  // ...
}
```

Dependencies

Contextual

Information that is needed to perform a task

```php
class CatApiConsumer {

  // ...

  public function getRandom() {

    // The endpoint path is required for this task to succeed.
    return $this->httpClient->get('/random', [
      // ...
      // The query parameters determine the results of the tas.
      'query' => [ 'type' => 'tabby' ],
    ]);
  }

}
```

Dependency

Configuration - Best Practices

Require Configuration dependencies in the class constructor

```
class CatApiConsumer {

  private $credentials;
  private $httpClient

  public function __construct(CredentialsInterface $credentials, HttpClientInterface $http_client) {
    $this->credentials = $credentials;
    $this->httpClient = $http_client;
  }

  // ...
}

$creds = new Credentials('daggerhart', 'abc123');
$client = new HttpClient(['base_url' => 'https://cat-pictures.api']);

$cat_pic_api = new CatApiConsumer($creds, $client);
```

Dependency

Contextual - Best Practices

Require Contextual dependencies in the method signature

```php
class CatApiConsumer {

  // ...

  // Now that the endpoint is a parameter we can rename the method to get()
  public function get(string $endpoint, array $query = []) {
    return $this->httpClient->get($endpoint, [
      'auth' => [
        $this->credentials->getUsername(),
        $this->credentials->getPassword()
      ],
      'query' => $query,
    ]);
  }
}
```

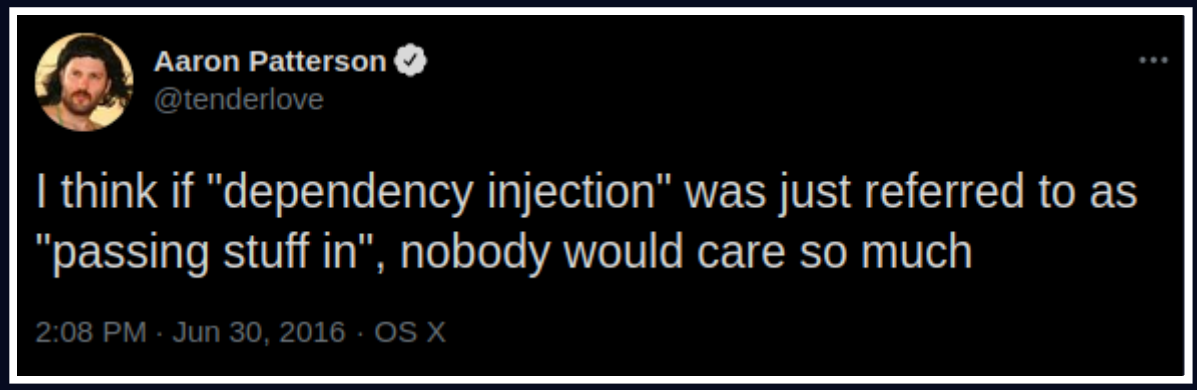
```php
$cat_pic_api = new CatApiConsumer($creds, $client);
$random_tabbies = $cat_pic_api->get('/random', [
  'type' => 'tabby',
]);
```

Dependencies - Best Practices

```php
class CatApiConsumer {

  private $credentials;
  private $httpClient

  public function __construct(CredentialsInterface $credentials, HttpClientInterface $http_client) {
    $this->credentials = $credentials;
    $this->httpClient = $http_client;
  }

  public function get($endpoint, $query = []) {
    return $this->httpClient->get($endpoint, [
      'auth' => [
        $this->credentials->getUsername(),
        $this->credentials->getPassword()
      ],
      'query' => $query,
    ]);
  }
}
```

```php
$cat_pic_api = new CatApiConsumer($creds, $client);
$random_tabbies = $cat_pic_api->get('/random', [
  'type' => 'tabby',
]);
```

Dependency Injection



https://twitter.com/tenderlove/status/748579020703313920

Passing stuff in

Dependencies - Best Practices

AKA: Dependency Injection

```php
class CatApiConsumer {

  private $credentials;
  private $httpClient

  public function __construct(CredentialsInterface $credentials, HttpClientInterface $http_client) {
    $this->credentials = $credentials;
    $this->httpClient = $http_client;
  }

  public function get(string $endpoint, array $query = []) {
    return $this->httpClient->get($endpoint, [
      'auth' => [
        $this->credentials->getUsername(),
        $this->credentials->getPassword()
      ],
      'query' => $query,
    ]);
  }
}
```

Patterns

A pattern is an approach to solving a common problem

Not a prescriptive set of classes or functions

Patterns

Everyday patterns in Drupal & Symfony

- Factory
- Service Locator / Container
- Decorator

Pattern - Factory

Deals with object creation mechanisms

Solves:

Creating complex objects reliably and easily

Pattern - Example

Simple Factory

Easily create complex objects

```
class SpecialtyPizzaFactory {

  public function createVeggie() {
    $pizza = new Pizza();
    $pizza->addTopping(new Topping('sauce'));
    $pizza->addTopping(new Topping('cheese'));
    $pizza->addTopping(new Topping('green pepper'));
    $pizza->addTopping(new Topping('onion'));
    $pizza->addTopping(new Topping('mushroom'));
    return $pizza;
  }

}
```

```
$factory = new SpecialtyPizzaFactory();
$veggie_pizza = $factory->createVeggie();
```

Pattern - Example

Method Factory

Static method on a class that creates instances of itself

```php
class CatApiConsumer {

  public function __construct(CredentialsInterface $credentials, HttpClientInterface $http_client) {
    $this->credentials = $credentials;
    $this->httpClient = $http_client;
  }

  public static function create() {
    return new static(
      new Credentials('daggerhart', 'abc123'),
      new HttpClient(['base_url' => 'https://cat-pictures.api'])
    );
  }
}
```

```php
$cat_api = CatApiConsumer::create();
$random_cats = $cat_api->get('/random');
```

Factory Summary

Factory services and methods allow us to easily instantiate complex objects fully prepared for use.

When creating new classes for Data Objects (Models). consider creating a Factory for that object. This allows other parts of the system to easily create an instance of the class.

Simple Factory Factory Method

Pattern - Service Locator

Stores instances of reusable classes that can be retrieved

```php
class ServiceLocator {

  private $services = [];

  public function add(string $name, $instance) {
    $this->services[$name] = $instance;
  }

  public function get(string $name) {
    return $this->services[$name];
  }
}
```

Allows other parts of the system to "Locate" services

```php
$service_locator = new ServiceLocator();
$service_locator->add('cat_api', new CatApiConsumer());
$service_locator->add('specialty_pizza_factory', new SpecialtyPizzaFactory());
// ...
$cat_api = $service_locator->get('cat_api');
```

Pattern - Service Locator

Drupal

Drupal has a Service Locator

```php
class Drupal {

  public static function service($id) {
    // ... container? ... interesting
    return static::getContainer()->get($id);
  }

}
```

Maybe you've seen this?

```php
\Drupal::database();
\Drupal::entityTypeManager();
\Drupal::config('my_custom_config');
\Drupal::service('some_contrib_module_service');
```

Pattern - Service Locator

Anti-Pattern

The Service Locator is a very useful concept, but is considered bad practice (anti-pattern) because it couples your code to the ServiceLocator object.

Consider:

```
class CatApiConsumer {

  private $httpClient;

  public function __construct() {
    $this->httpClient = \Drupal::service('http_client');
  }

}
```

Pattern - Service Locator

Improvements

Don't use the \Drupal Service Locator within classes

```
class CatApiConsumer {

  private $httpClient;

  public function __construct(HttpClientInterface $http_client) {
    $this->httpClient = $http_client;
  }

}

$http_client = \Drupal::service('http_client');
$cat_api = new CatApiConsumer($http_client);
```

Pattern - Service Locator

Improvements cont...

Use a Method Factory and pass the Service Locator into it.

```
class CatApiConsumer {

  private $httpClient;

  public function __construct(HttpClientInterface $http_client) {
    $this->httpClient = $http_client;
  }

  // Factory!
  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('http_client')
    );
  }
}
```

```
$container = \Drupal::getContainer();
$cat_api = CatApiConsumer::create( $container );
```

Pattern - ~~Service Locator~~

Dependency Injection Container

When we use a Service Locator in a way that removes coupling between our classes and the Service Locator, this is called a "Dependency Injection Container"

Pattern - DI Container

Notice: We didn't change how the Service Locator worked, we changed how we use it.

```
class CatApiConsumer {

  private $httpClient;

  public function __construct(HttpClientInterface $http_client) {
    $this->httpClient = $http_client;
  }

  // Factory!
  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('http_client')
    );
  }
}
```

```
$container = \Drupal::getContainer();
$cat_api = CatApiConsumer::create( $container );
```

Pattern - DI Container

Best Practices

Don't use the Service Locator directly:

```
class CatApiConsumer {
  public function __construct() {
    $this->httpClient = \Drupal::service('http_client');
  }
}
```

Use the Dependency Injection Container in Factories.

```
class CatApiConsumer {
  public function __construct(HttpClientInterface $http_client) {
    $this->httpClient = $http_client;
  }

  public static function create(ContainerInterface $container) {
    return new static( $container->get('http_client') );
  }
}
```

Service Locator Summary

- Service Locator stores reusable objects
- Accessing the Service Locator directly couples your code to it
- When we pass around a Service Locator to Factories it becomes a Dependency Injection Container

Service Locator

Dependency Injection Containers:

- Are Service Locators that are passed around rather than accessed directly
- Should be passed into Factories
- ... and more

Services Dependency Injection

Drupal

Define your service dependencies as "arguments"

```
services:
  my_service_name:
    class: \Drupal\my_module\MyCustomService
    arguments:
      - '@entity_type_manager'
      - '@some_other_service'
```

The container will inject dependencies into the service's constructor

```
class MyCustomService {

  public function __construct(EntityTypeManagerInterface $entity_type_manager, $other_service) {}

}
```

Pattern - Decorator

Data Objects

Dynamically adding new functionality (behavior) to a class through composition

```
class TaxedPrice {

  private $product;

  public function __construct(ProductInterface $product) {
    $this->product = $product;
  }

  public function getTaxedPrice() {
    return $this->product->getPrice() * 1.07;
  }

  public function getProduct() {
    return $this->product;
  }

}
```

Pattern - Decorator

Allows us to separate the concerns of objects

```
class Cat {
  private $type = 'tabby';

  public function getType() {
    return $this->type;
  }
}
```

```
class CatPictureDecorator {
  private $cat;
  private $catApi;

  public function __construct(CatInterface $cat, CatApiConsumerInterface $cat_api) {
    $this->cat;
    $this->catApi = $cat_api;
  }

  public function getPicture() {
    return $this->catApi->get('/random', ['type' => $this->cat->getType()]);
  }
}
```

Pattern - Decorator

In Drupal

Consider:

- We have a content type (node type) named "Cat"
- Since it is a Node, it is already very complicated through multiple levels of inheritance
- A Node instance already has state that may be modified

```
function hook_node_view($build, $entity, $display, $view_mode) {
  $build['cat_picture'] = [
    '#markup' => // ???
  ];
}
```

How should we extend the functionality of a Cat node?

Pattern - Decorator

Data Object Decorators in Drupal

With inheritance, we have to re-instantiate all our nodes as "Cat"s

```
class Node extends EditorialContentEntityBase implements NodeInterface {

  public function __construct($values, $entity_type, $bundle, $translations) {}
}
```

```
class Cat extends Node {

  public function getPicture() {
    $cat_api = \Drupal::service('cat_api');
    return $cat_api->get('/random', ['type' => $this->field_cat_type->value]);
  }
}
```

```
function hook_node_view($build, $entity, $display, $view_mode) {
  // How are we going to get an instance of this node as a Cat?
  $build['cat_picture'] = [
    '#markup' => // ???
  ];
}
```

Pattern - Decorator

With a Decorator, we can use existing instances of the node

```
class CatPictureDecorator {
  private $node;
  private $catApi;

  public function __construct($node, $cat_api) {
    $this->node;
    $this->catApi = $cat_api;
  }

  public function getPicture() {
    $query = ['type' => $this->node->field_cat_type->value];
    return $this->catApi->get('/random', $query);
  }
}
```

```
function hook_node_view($build, $entity, $display, $view_mode) {
  $cat_api = \Drupal::service('cat_api');
  $cat = new CatPictureDecorator($entity, $cat_api);
  $build['cat_picture'] = [
    '#markup' => "<img src='{$cat->getPicture()}'>",
  ];
}
```

Decorator Summary

The Decorator pattern allows us to:

- extend functionality of existing objects (in current state)
- separate the concerns of object functionality
- use existing instances of the object without reloading

Decorator Pattern

Service Decorators

Heads up!

Symfony provides a mechanism for decorating services

```
services:
  # Some service
  password_generator_unambiguous:
    class: \Drupal\services_examples\PasswordGeneratorUnambiguous

  # Example service decorator
  password_generator_unambiguous_decoration:
    class: \Drupal\services_examples\PasswordGeneratorUnambiguousDecoration
    public: false
    decorates: password_generator_unambiguous
    arguments:
      - '@password_generator_unambiguous_decoration.inner'
```

This has different considerations than a simple data object decorator, as it must provide all the functionality that the decorated service offers.

Drupal

Let's build a custom module!

- Data Objects
- Services
- Dependency Injection
- Decorators

Drupal Module

Planning

The Cat API module uses a 3rd party API for gather cat information

https://docs.thecatapi.com/

```
name: Cat Api
description: Retrieves data from the Cats API
type: module
core_version_requirement: ^8.8 || ^9
package: Custom
configure: cats_api.settings_form
```

Cat API Module

Features

- Custom service for consuming the The Cat Api
  - Factory for creating our service
- Configuration Form for service credentials
- Custom block type for displaying random cat pictures
- Custom route for browsing The Cat Api

github.com/daggerhart/drupal8_examples

Cat API - Settings

config/schema/cat_api.schema.yml

```yaml
cat_api.settings:
  type: config_object
  mapping:
    base_uri:
      type: string
    api_key:
      type: string
```

config/schema/cat_api.settings.yml

```yaml
base_uri: 'https://api.thecatapi.com/v1/'
api_key: ''
```

Implication

```php
$cat_api_settings = \Drupal::config('cat_api.settings');
$cat_api_settings->get('base_uri');
$cat_api_settings->get('api_key');
```

Cat API

Client Service Interface

```php
namespace Drupal\cat_api\Service;

/**
 * Interface CatApiClientInterface.
 *
 * @package Drupal\cat_api\Service
 */
interface CatApiClientInterface {

  /**
   * Retrieve data from the cat api.
   *
   * @param string $endpoint
   * @param array $query
   *
   * @return array
   */
  public function get(string $endpoint, array $query = []): array;

}
```

Cat API - Client Service

```php
namespace Drupal\cat_api\Service;

use Drupal\Component\Serialization\Json;
use GuzzleHttp\Client;

class CatApiClient implements CatApiClientInterface {

  private $httpClient;
  private $json;

  public function __construct(Client $http_client, Json $json) {
    $this->httpClient = $http_client;
    $this->json = $json;
  }

  public function get(string $endpoint, array $query = []) {
    $response = $this->httpClient->get($endpoint, [
      'query' => $query,
    ]);

    return $this->json::decode($response->getBody()->getContents());
  }
}
```

Cat API - Factory Service

```php
namespace Drupal\cat_api\Service;

class CatApiClientFactory {
  private $configFactory;
  private $httpClientFactory;
  private $json;

  public function __construct($config_factory, $http_client_factory, $json) {
    $this->configFactory = $config_factory;
    $this->httpClientFactory = $http_client_factory;
    $this->json = $json;
  }

  public function create() {
    $config = $this->configFactory->get('cat_api.settings');
    $http_client = $this->httpClientFactory->fromOptions([
      'base_uri' => $config->get('base_uri'),
      'headers' => [
        'x-api-key' => $config->get('api_key'),
      ],
    ]);

    return new CatApiClient($http_client, $this->json);
  }
}
```

Cat API - Factory Service

cat_api.services.yml

```yaml
services:
  # Client factory service
  cat_api.client_factory:
    class: \Drupal\cat_api\Service\CatApiClientFactory
    arguments:
      - '@config.factory'
      - '@http_client_factory'
      - '@serialization.json'
  # ...
```

Remember:

```php
class CatApiClientFactory {

  public function __construct($config_factory, $http_client_factory, $json) {
    $this->configFactory = $config_factory;
    $this->httpClientFactory = $http_client_factory;
    $this->json = $json;
  }
  // ...
}
```

Cat API - Client Service

```
services:
  # ...
  # Create a service from another Service Factory
  cat_api.client:
    class: \Drupal\cat_api\Service\CatApiClient
    factory: ['@cat_api.client_factory', 'create']
```

Remember:

```
class CatApiClientFactory {
  // ...

  public function create() {
    $config = $this->configFactory->get('cat_api.settings');
    $http_client = $this->httpClientFactory->fromOptions([
      'base_uri' => $config->get('base_uri'),
      'headers' => [
        'x-api-key' => $config->get('api_key'),
      ],
    ]);

    return new CatApiClient($http_client, $this->json);
  }
}
```

Cat API - Services Yaml

Complete

```yaml
services:
  # Client factory service
  cat_api.client_factory:
    class: \Drupal\cat_api\Service\CatApiClientFactory
    arguments:
      - '@config.factory'
      - '@http_client_factory'
      - '@serialization.json'

  # Create a service from another Service Factory
  cat_api.client:
    class: \Drupal\cat_api\Service\CatApiClient
    factory: ['@cat_api.client_factory', 'create']
```

- When someone request the Factory from the container, it has its dependencies injected
- When someone request the Client from the container, it is build by the Factory

Service Factories

Drupal Core Examples

`web/core/core.services.yml`

Guzzle Http Client

```yaml
# Factory
http_client_factory:
  class: Drupal\Core\Http\ClientFactory
  arguments: ['@http_handler_stack']

# Generic http client instance.
http_client:
  class: GuzzleHttp\Client
  factory: ['@http_client_factory', 'fromOptions']
```

Events - Services

Since Symfony events are service classes, we can inject dependencies are arguments in our service definition

```
services:
  my_custom_event:
    class: \Drupal\my_module\MyCustomEvent
    arguments:
      - '@config.factory'
      - '@entity_type_manager'
    tags:
      - { name: 'event_subscriber' }
```

```
namespace Drupal\my_module;

class MyCustomEvent implements EventSubscriberInterface {
  private $config;
  private $entityTypeManager;

  public function __construct($config_factory, $entity_type_manager) {
    $this->config = $config_factory->get('my_module.settings');
    $this->entityTypeManager = $entity_type_manager;
  }

  // ...
}
```

Dependency Injection

in Drupalisms

- Controllers (Route Handlers)
- Custom Forms
- Custom Block Type (Plugins)

How do we inject dependencies when we don't control the instantiation of a class?

Container Injection Interface

```php
namespace Drupal\Core\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Defines a common interface for dependency container injection.
 *
 * This interface gives classes who need services a factory method for
 * instantiation rather than defining a new service.
 */
interface ContainerInjectionInterface {

  /**
   * Instantiates a new instance of this class.
   *
   * This is a factory method that returns a new instance of this class. The
   * factory should pass any needed dependencies into the constructor of this
   * class, but not the container itself. Every call to this method must return
   * a new instance of this class; that is, it may not implement a singleton.
   *
   * @param \Symfony\Component\DependencyInjection\ContainerInterface $container
   *   The service container this instance should use.
   */
  public static function create(ContainerInterface $container);

}
```

Controller with DI

By implementing the `ContainerInjectionInterface`,
Drupal will instantiate our controller using the create() method factory

```
# cat_api.routing.yml
cat_api.browse_cats_page:
  path: '/browse-cats'
  defaults:
    _controller: '\Drupal\cat_api\Controller\BrowseCatsPage::page'
  # ...
```

```
namespace Drupal\cat_api\Controller;

class BrowseCatsPage implements ContainerInjectionInterface {

  public function __construct(CatApiClientInterface $cat_api_client) {
    $this->catApiClient = $cat_api_client;
  }

  public function page() {}

  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('cat_api.client')
    );
  }
}
```

Form with DI

Custom forms in Drupal `extend` the `FormBase` class...

```php
namespace Drupal\cat_api\Form;

use Drupal\Core\Form\FormBase;

class SearchCatsForm extends FormBase {}
```

... and `FormBase` implements `ContainerInjectionInterface`

```php
namespace Drupal\Core\Form;

// use ...

abstract class FormBase implements FormInterface, ContainerInjectionInterface {}
```

Form with DI

therefore...

our custom forms can have a method factory named `create()`

```php
namespace Drupal\cat_api\Form;

use Drupal\Core\Form\FormBase;
use Symfony\Component\DependencyInjection\ContainerInterface;

class SearchCatsForm extends FormBase {
  private $catApiClient;

  public function __construct(CatApiClientInterface $cat_api_client) {
    $this->catApiClient = $cat_api_client;
  }

  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('cat_api.client')
    );
  }

  // ...
}
```

Plugins w/ DI

Plugins do not inherit an interface that allows for Dependency Injection by default, but there is an interface for that: `ContainerFactoryPluginInterface`

```php
namespace Drupal\Core\Plugin;

use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Defines an interface for pulling plugin dependencies from the container.
 */
interface ContainerFactoryPluginInterface {

  public static function create(
    ContainerInterface $container,
    array $configuration,
    $plugin_id,
    $plugin_definition
  );
}
```

This one is a little trickier, because the plugin constructor has many default requirements

Plugins w/ DI - Example

```php
/**
 * @Block(
 *   id = "cat_api_random_cat",
 *   admin_label = @Translation("Cat API - Random Cat")
 * )
 */
class CatApiRandomCat extends BlockBase implements ContainerFactoryPluginInterface {

  private $catApiClient;

  public function __construct($configuration, $id, $definition, $cat_api_client) {
    parent::__construct($configuration, $plugin_id, $plugin_definition);
    $this->catApiClient = $cat_api_client;
  }

  public static function create($container, $configuration, $id, $definition) {
    return new static(
      $configuration,
      $id,
      $definition,
      $container->get('cat_api.client')
    );
  }
}
```

Dependency Injection in Drupal

Takeaways

There is a way to do that!

- Services
  - Specify dependencies as arguments in *.services.yml
- Controllers (Route Handlers)
  - Implement ContainerInjectionInterface yourself
- Custom Form
  - FormBase already implements ContainerInjectionInterface
- Custom Block Type (Plugins)
  - Implement ContainerFactoryPluginInterface yourself

Questions?